Open Source Tooling

Construyendo herramientas que la gente realmente usa

ZACH "LEIF" ERIKSEN

Copyright

Gratuito para leer en línea. El ePub es de pago libre; si te fue útil, puedes apoyar el trabajo.

github.com/0xLeif · leif.algo

Uno de cuatro libros del conjunto agent-stack. Cómo fue hecho está en el colofón al final.

Dedicatoria

Para todos los que construyen en abierto, y lo publican de todas formas.

La Biblioteca

Estos libros se sostienen solos, pero fueron escritos como un conjunto. El código se abarató y la confianza escaseó. Juntos forman un solo argumento: qué construir ahora, y cómo confiar en ello.

- **The Agent Developer's Field Guide:** Construyendo herramientas, especificaciones y confianza para agentes que publican código real
- **First-Class:** Construyendo para humanos y agentes por igual
- **Building Agents:** Notas del intento de dar manos propias al software
- **Open Source Tooling:** Construyendo herramientas que la gente realmente usa *(este libro)*

Gratuito para leer en línea. Cada ePub es de pago libre.

Contenido

Introducción

Este libro trata sobre el arte de construir herramientas que otras personas, y otros agentes, realmente usan.

Gira en torno a fledge, una sola herramienta de línea de comandos que cubre todo el ciclo de vida del desarrollo, y la pregunta que me obligó a responder una y otra vez: ¿qué hace que una herramienta valga la pena como dependencia? La tesis es que la respuesta es la misma para un humano que para un agente. Una buena herramienta tiene una superficie limpia, sin pasos interactivos ocultos, un límite de plugin real, y una especificación que se mantiene honesta a medida que el código cambia. Constrúyela así, y un agente puede manejarla desde el primer día, porque nunca fue diseñada para necesitar una persona para empezar.

Este es el segundo de los dos libros de evidencia. *First-Class* argumenta que el software debe ser de primera clase para ambos. La *Field Guide* convierte eso en un método. *Building Agents* muestra los agentes. Este muestra las herramientas que los sustentan, hasta el fondo: por qué Rust fue la elección correcta y cómo el sandbox de WASM evita que un plugin haga algo que no debe.

Es para cualquiera que alguna vez haya publicado una herramienta y la haya visto usarse de formas que no planeó. No necesitas fledge. Necesitas los hábitos que hacen que una herramienta sobreviva el contacto con usuarios que no son tú, incluidos los que ni siquiera son humanos.

Una CLI para todo el ciclo de vida

Cada repositorio tenía un dialecto diferente. Makefiles distintos, scripts distintos, READMEs distintos, cada uno con su propia idea de cómo se construye, prueba y ejecuta la cosa. Nada transfería. Abrías un proyecto que no habías tocado en un tiempo y el primer trabajo era descifrar el conjuro local de nuevo. Qué script, qué objetivo, qué orden. El trabajo en sí no era difícil. Ejecutar pruebas no es difícil. El problema era que era diferente en todos lados, y la diferencia era pura sobrecarga.

Así que quería una interfaz consistente en todos ellos. Ahí es donde comienza fledge. La descripción en el repositorio lo dice sin rodeos: *una CLI, todo tu ciclo de vida de desarrollo.*

Había tres cosas que me empujaron hacia allí.

La primera era ese problema de uniformidad, cada repositorio hablando su propio idioma. La segunda era el arranque. Seguía copiando y pegando la misma configuración e inicialización en cada proyecto nuevo. El mismo montón de archivos, el mismo cableado, una y otra vez, antes de poder empezar con lo que realmente quería construir. La tercera era la escala. Pongo en marcha toneladas de proyectos, con agentes trabajando en ellos, y necesitaba una sola CLI en la que todos pudieran confiar. No "una herramienta que uso". Una herramienta en la que todo y todos los que trabajan en esos repositorios pudieran apoyarse de la misma manera.

Ese último punto es fácil de perder. fledge no nace de un repositorio que me molestó. Nace de tener muchos repositorios, muchos proyectos en marcha, y agentes en el ciclo de todos ellos, y necesitar que todo eso hablara con una superficie consistente en lugar de cien superficies a medida. El ángulo de que-los-agentes- pueden-depender-de-una-sola-CLI es su propio hilo, y lo desarrollo en el libro de agentes. Consistencia, scaffolding y escala: ese es el origen.

Construido para humanos y agentes por igual

Hay una idea detrás de todo esto, y es más grande que fledge. Gran parte de mis herramientas surge de creer que humanos y agentes van a usar las mismas herramientas.

Ahora mismo, la mayoría de lo que existe es primero-para-humanos. Los proyectos se construyen para personas, y luego intentamos incorporar agentes después del hecho.

Puede haber cosas primero-para-agentes y cosas primero-para-humanos, pero lo que realmente necesitamos es hacer proyectos primero-para-agentes-y-humanos, construidos desde el inicio para que ambos sean de primera clase.

Eso son todas mis herramientas. Una herramienta debe funcionar de primera clase de cualquier manera: un humano puede usarla sin un agente, y un agente puede usarla sin un humano. Ninguno es la ocurrencia de último momento. Y cuando un agente la usa, la herramienta debe *ayudarlo*, no dejarlo adivinando todos los comandos y cómo funciona todo.

Diseñar para humanos y agentes como usuarios de primera clase igualitarios es la razón real por la que fledge tiene el aspecto que tiene, y es lo que hay que tener en mente para el resto de este libro.

Qué significa realmente cero configuración

Cuando digo cero configuración, me refiero a que no tienes que enseñarle a la herramienta sobre tu proyecto antes de que pueda ayudarte. La dejas caer en un repositorio y funciona. Hay tres piezas para eso.

Detecta automáticamente el proyecto y sus comandos. Swift, Rust, Node, lo que sea. Descifra qué tipo de proyecto tiene delante y conoce el build/test/run correcto para él. Sin paso de configuración donde primero describes tu proyecto a un archivo de config.

Los mismos verbos funcionan en todos los proyectos. `build`, `test`, `run`, `lint`, las mismas palabras, independientemente de lo que haya debajo. Ese es el beneficio directo del problema de "cada repositorio tenía un dialecto diferente". Los dialectos siguen ahí abajo; Cargo sigue siendo Cargo y Swift Package Manager sigue siendo Swift Package Manager. Pero dejas de tener que preocuparte ante cuál estás parado. Aprendes los verbos una vez y son los verbos en todas partes. La detección maneja la traducción hacia cualquier herramienta subyacente real.

Esta es la idea de humanos-y-agentes hecha concreta. Una persona deja de tener que volver a aprender cada repositorio. Un agente deja de tener que *adivinar*. No tiene que ir a buscar el comando correcto para este proyecto en particular, ni leer el README con la esperanza de encontrarlo. Ejecuta `build`, ejecuta `test`, y la herramienta ya sabe qué significa eso aquí. Lo que me ahorra el reaprendizaje es lo mismo que evita que el agente adivine. fledge fue construido primero para mí y los agentes que ejecuto, y se gana su lugar ahí antes que en cualquier otro sitio.

Y se extiende mediante plugins. Un repositorio nuevo obtiene el núcleo, y se añaden capacidades incorporando plugins. El núcleo conoce los verbos comunes y cómo detectar los tipos de proyectos comunes; todo lo demás viene de los plugins. Así que cero configuración no significa "hace todo de entrada". Significa que la parte que obtienes de entrada no necesita configuración, y creces desde ahí añadiendo plugins, no escribiendo config.

Concretamente, una primera ejecución va así. Entras en un repositorio y ejecutas `fledge`, y detecta automáticamente el proyecto (Swift, Rust, Node, lo que sea) y simplemente conoce el build/test/run correcto para él, sin paso de config. Pídele que introspeccione y te *dice los verbos disponibles* para este repositorio: se autodescribe, así que tú (o un agente) no tiene que adivinar qué es posible. Si es un proyecto completamente nuevo, el primer movimiento real suele ser el scaffolding desde una plantilla en lugar de detectar uno existente. Y todo funciona sin cabeza desde el primer comando: configura `FLEDGE_NON_INTERACTIVE`, pide `--json`, y un agente lo maneja desde el paso uno. Detectar, introspeccionar, quizás scaffoldear, y luego ejecutar.

La parte del ciclo de vida

El repositorio llama a fledge un ejecutor de tareas de cero configuración con scaffolding y revisión con IA. La ejecución de tareas es el conjunto de verbos anterior: el build/test/run/lint del día a día. El scaffolding es la respuesta al segundo problema de origen: en lugar de copiar y pegar la misma configuración en cada proyecto nuevo a mano, fledge lo levanta. Y también hay una pieza de revisión con IA integrada en el ciclo de vida.

La ejecución de tareas vino primero. Es la semilla, la parte sobre la que creció todo lo demás. El scaffolding son realmente plantillas: levantar un proyecto nuevo desde una en lugar de copiarlo y pegarlo. Y la revisión con IA se incorporó porque la revisión es parte del ciclo de desarrollo. Si los agentes están escribiendo el código, calificarlo no es un ritual separado que vas a ejecutar en otro lugar, es solo otro verbo junto a build y test. Una superficie le gana a tres herramientas, para mí y aún más para un agente que de otro modo tendría que aprender tres cosas. Estos tres son los pilares estructurales, el núcleo, con los plugins como capa de extensión a su alrededor.

El eslogan de fledge es "una CLI, todo tu ciclo de vida de desarrollo, ejecutor de tareas de cero configuración, scaffolding de proyectos, revisión con IA, y más", y eso es exactamente las tres piezas. Es un único binario en Rust, y lo obtienes de la manera obvia y aburrida: `brew install CorvidLabs/tap/fledge` desde el tap de

Homebrew, o `cargo install fledge` si lo prefieres, o un script de shell. Nada exótico para instalarlo.

Cómo lo uso realmente

Esta no es una herramienta a la que recurro a veces. Es todos los repositorios, todo el tiempo, la forma predeterminada en que construyo, pruebo y ejecuto todo ahora. Y mis agentes la manejan. Los agentes ejecutan fledge más de lo que yo lo hago manualmente. Se construyó para domar muchos repositorios, y ahora es la única CLI en la que tanto yo como los agentes que tengo trabajando en esos repositorios confiamos, en todas partes.

Eso también es por lo que importa la siguiente pieza. Una CLI en la que todos se apoyan, que cualquiera puede extender, que los agentes instalan y ejecutan, no puede ser un conjunto fijo de características que yo apruebo una a la vez. Tiene que ser extensible por cualquiera, en el lenguaje que quieran, sin tocar el núcleo. Y una vez que dejas entrar plugins arbitrarios, y una vez que los agentes son los que los ejecutan, tienes que pensar seriamente en la confianza. Ese es el siguiente capítulo.

Por qué fledge y no Make

La gente pregunta esto en cuanto escucha lo que hace fledge. ¿Construiste un ejecutor de tareas? Está Make. Está Just. Está Turborepo. Está el bloque de scripts de npm ahí mismo en cada package.json. ¿Por qué escribir otro?

La respuesta honesta es que el muro contra el que choqué nunca fue Make. Empecé como desarrollador de iOS, y para automatizar una compilación allí el camino siempre llevaba al mismo lugar: fastlane, que significa Ruby. No quería Ruby. Soy una persona de Swift. Quería automatizar mis compilaciones en el lenguaje en el que trabajo, y en cambio todo el ecosistema me canalizaba hacia un DSL de Ruby y un `Gemfile` y una herramienta que era su propio pequeño mundo que aprender. Cada vez que quería publicar algo la respuesta era "escribe un lane de fastlane", y cada lane de fastlane era Ruby que no quería escribir, en un runtime que no quería gestionar, haciendo un trabajo que sentía que debería poder hacer en Swift. De ahí creció el dolor de fledge. No de "Make es torpe". Era "¿por qué me obligan al lenguaje de otro para automatizar mi propio proyecto?"

Así que cuando la gente pone fledge al lado de Make y Just, apuntan al objetivo equivocado. La respuesta no es que esas herramientas sean malas. Lo que quería de un ejecutor de tareas eran tres cosas, y el muro de fastlane es de donde vienen las tres.

Lo quería a mi manera, en mi lenguaje, en mis términos, sin quedar encadenado a un runtime de la forma en que fastlane te encadena a Ruby. Un ejecutor de tareas es la cosa que tocas cien veces al día, y con el tiempo dejas de querer vivir dentro de las elecciones de otro sobre cómo debe sentirse, o en qué lenguaje tienes que pensar para usarlo. Quería que `build`, `test` y `run` significaran lo mismo en cada repositorio, y quería que los pasos debajo fueran escribibles en cualquier lenguaje que requiriera el trabajo en lugar de ser empujado hacia Ruby. Make no te lo impide, pero Make tampoco te lo da. Construyes la consistencia tú mismo, en cada Makefile, a mano, para siempre. Una herramienta que me permite reinventar el dialecto por proyecto no ha resuelto mi problema; solo me ha dado un lugar más agradable para seguir reinventándolo.

Las otras dos son las del primer capítulo, y el archivo de fastlane es donde también las habría querido. Ninguna de esas herramientas es primero-para-agentes (JSON por defecto, introspección, un modo sin cabeza), que es la parte que más me importa

ahora que mis agentes manejan esto más de lo que yo lo hago manualmente. Y ninguna de ellas es un único binario ligero que cae en un repositorio sin runtime que instalar primero, que es exactamente lo que tiene que ser una superficie consistente en un montón de repositorios de diferentes lenguajes. Un lane de fastlane necesita Ruby; los scripts de npm necesitan npm; un ejecutor que escribiría en TypeScript necesita un runtime de TypeScript. fledge no necesita nada.

Nada de lo cual es "Make es malo". Make es genial en lo que Make hace, Just es un ejecutor de comandos limpio, Turborepo cachea bien un monorepo de JS. Si uno de esos es todo lo que necesitas, úsalo. No voy a pretender que fledge gana una pelea característica a característica en su propio terreno.

Pero ninguno de ellos era la cosa que realmente habría querido estando frente a un archivo de fastlane años antes: una superficie, primero-para-agentes, un único binario ligero, y la libertad de escribir los pasos en cualquier lenguaje que encajara en lugar de ser canalizado hacia Ruby. fledge es la herramienta que desearía haber tenido entonces, finalmente construida.

---

fledge y MCP

MCP es el estándar ambiental ahora. En 2026, los agentes consumen herramientas principalmente a través de servidores MCP. Si construyes una herramienta y quieres que los agentes puedan usarla de manera confiable, el camino de menor resistencia es exponer un servidor MCP. Ese es simplemente el estado del ecosistema.

fledge no tiene un servidor MCP. Y sin embargo, los agentes lo manejan constantemente en docenas de repositorios, y funciona. Vale la pena ser explícito sobre la razón, porque dice algo sobre qué pieza construir primero.

La CLI es el primitivo

Un servidor MCP es una interfaz de producción. Maneja el enrutamiento de solicitudes, registra logs, te da observabilidad estructurada sobre lo que un agente pidió y lo que obtuvo de vuelta. Esas son cosas buenas de tener. Pero un servidor MCP es una capa que pones encima de algo. La pregunta es: ¿encima de qué?

Si construyes el servidor MCP primero y la CLI es una ocurrencia de último momento, obtienes una herramienta que funciona para agentes y es un dolor para humanos. Si construyes la CLI primero, obtienes una herramienta que funciona para agentes ahora mismo sin adaptador de protocolo, funciona para humanos, y puede tener un servidor MCP puesto delante cuando lo necesites. La CLI es el primitivo. Todo lo demás se asienta sobre ella.

fledge fue construido con eso en mente. Cada comando emite `--json`. Existe `FLEDGE_NON_INTERACTIVE` para la ejecución sin cabeza. `fledge introspect` le da a cualquier llamador, humano o agente, un manifiesto estructurado de cada verbo disponible, qué hace, y qué acepta. No hay prompts interactivos que bloqueen ejecuciones desatendidas. La herramienta se autodescribe.

Ese perfil, salida estructurada por defecto, un manifiesto de capacidades explícito, sin pasos interactivos ocultos, es exactamente lo que una superficie de herramientas MCP le da a un agente. fledge tiene todo eso sin el protocolo, porque esas propiedades vinieron de diseñar para llamadores agentes desde el principio, no de envolver la herramienta después para hacerla amigable para agentes.

Una instancia de Claude que maneja fledge hoy llama a `fledge introspect`, recibe de vuelta un manifiesto JSON de lo que está disponible, elige el verbo correcto, pasa --

`json`, y lee la salida estructurada. Ese es el mismo ciclo de interacción que un servidor MCP mediaría, menos el encuadre MCP. La CLI ya es la superficie de herramienta limpia.

MCP no es el competidor

El encuadre que a veces surge, "¿CLI o MCP?", los trata como alternativas. No lo son. MCP es una especificación de transporte y protocolo. La CLI es la que hace el trabajo. La pregunta no es cuál tener; es cuál construir primero y cuál poner encima como capa.

Construye primero la CLI, del tipo que se describe a si misma, emite JSON y corre sin cabeza. Una vez que tienes eso, exponer un servidor MCP encima es sencillo: mapeas cada entrada de `fledge introspect` a una definición de herramienta MCP, llamas a la CLI por debajo, devuelves la salida JSON. El núcleo ya es JSON estructurado sobre un límite limpio con un manifiesto de capacidades explícito. Un servidor MCP es una API de producción encima de ese mismo primitivo, de la misma manera que una API REST podría estar delante de una librería bien estructurada. La interfaz cambia; el trabajo no.

Y porque la CLI es la superficie real, todo lo que puede llamar a un subproceso puede usarla sin el adaptador de protocolo. Un script de shell puede usarla. Un ejecutor de CI puede usarla. Un humano en una terminal puede usarla. Un cliente MCP puede usarla a través de la capa del servidor. Ninguno de esos llamadores requiere que los otros existan. Obtienes universalidad del primitivo, no del protocolo.

Lo que MCP agrega

Nada de esto es un argumento en contra de MCP. Agrega cosas reales una vez que esta delante de la CLI.

Logging y observabilidad. Un servidor MCP se sienta entre el agente y la herramienta, así que puedes registrar cada llamada, cronometrarla, inspeccionar argumentos, marcar anomalías. La CLI invocada directamente te da lo que sea que canalices a un archivo de log. La capa MCP te da observabilidad estructurada sin instrumentar cada comando individualmente. Para uso en producción donde quieres auditar lo que pidió un agente, eso importa.

Descubribilidad al nivel del protocolo. MCP tiene una forma estandarizada de que un agente pregunte "¿qué herramientas hay aquí?" y reciba de vuelta una respuesta estructurada. fledge tiene `fledge introspect` para lo mismo, pero `fledge introspect`

requiere saber que fledge está ahí. Un registro MCP le permite a un agente descubrir la herramienta a través de un handshake estándar antes de que sepa nada sobre la herramienta que hay debajo.

Composición entre herramientas. Un servidor MCP puede exponer varias herramientas subyacentes a través de un solo punto de acceso, así que un agente tiene un lugar al que conectarse en lugar de conocer una lista de CLIs individuales. Esa es una conveniencia operacional cuando el número de herramientas es grande.

Esos son argumentos de infraestructura de producción. Se aplican cuando estás ejecutando agentes a escala, auditando su comportamiento, y conectándolos a una amplia superficie de herramientas a través de una capa gestionada. El servidor MCP es, como etiqueta de lo que es, una API orientada a IA con logging. Eso es algo útil de tener. Simplemente no es lo que construyes primero.

El orden de las operaciones

CLI primero. Funciona para cada llamador en el momento en que existe. El servidor MCP es el envoltorio que agregas cuando el logging y la estandarización de protocolo valen la capa, no antes.

Para fledge, la CLI es la base sobre la que se asientan un humano ejecutando comandos, un agente manejando un repositorio, y un servidor MCP hablando con clientes downstream. La historia de los agentes no es "los agentes usan fledge a través de MCP". Es "los agentes usan fledge de la misma manera que lo hace todo lo demás, porque la CLI fue construida de primera clase para cualquier llamador."

Cuando exista un servidor MCP para fledge, será una capa delgada. El trabajo ya está hecho en el núcleo. Ese es el punto de construir el primitivo primero.

Scaffolding y plantillas

La segunda cosa que me empujó hacia fledge, después del problema de uniformidad, fue el arranque: el dolor de copiar-y-pegar-la-misma-configuración que nombré en el capítulo uno. Esto es lo que parece de cerca. Decides hacer una pequeña CLI en Rust y la primera hora no es la CLI. Es el diseño de Cargo, la configuración, la configuración de lint, el CI, el esqueleto del README, toda la cosa que ya has escrito veinte veces. Ese es el impuesto, y lo estaba pagando constantemente porque pongo en marcha muchos proyectos.

Así que el scaffolding es un pilar, no una característica secundaria. Levantar un proyecto desde cero es parte del ciclo de vida tanto como construirlo y probarlo. La idea completa es: levantar un nuevo proyecto desde una plantilla en lugar de copiarlo y pegarlo a mano.

En fledge eso vive bajo `fledge templates`. Ejecutas `fledge templates init` con un nombre y una plantilla y te levanta el proyecto:

```
fledge templates init my-tool --template rust-cli
```

Hay un conjunto integrado que cubre los tipos de proyectos que realmente inicio. Los que se publican son `rust-cli`, `ts-bun`, `python-cli`, `go-cli`, `ts-node`, `static-site`, `kotlin-kmp`, y `kotlin-ktor-api`. Esa lista es básicamente un mapa de los lenguajes en los que trabajo: una CLI en Rust, un par de sabores de TypeScript, una CLI en Python, una CLI en Go, un sitio estático, y los de Kotlin Multiplatform y Ktor API. Esos ocho son el conjunto integrado completo en el repositorio hoy. El lado de detección de fledge sabe cómo manejar Swift, Rust, Node, y el resto una vez que existe un proyecto; el lado de plantillas es cómo un proyecto llega a existir en primer lugar.

La parte que más me importa es que las plantillas no son una lista cerrada que tengo que aprobar una a la vez. Puedes hacer scaffolding desde cualquier repositorio de GitHub, no solo desde los integrados. `--template usuario/repo` y lo extrae de ahí:

```
fledge templates init my-app --template user/repo
```

El núcleo publica un conjunto pequeño y útil, y más allá de eso lo extiendes tú mismo sin que yo tenga que estar en el ciclo. Una plantilla es solo una forma de proyecto que alguien ya resolvió, y si vive en un repositorio de GitHub, fledge puede levantar

un nuevo proyecto desde ella. Así que el conjunto de plantillas no es "lo que CorvidLabs publicó". Es "cada punto de partida que alguien haya puesto alguna vez en un repositorio."

Y hay un conjunto completo de verbos alrededor de las plantillas, no solo `init`. Está `fledge templates create` para crear una, `fledge templates list` y `fledge templates search` para encontrarlas, y `fledge templates validate` para verificar que una plantilla está bien formada antes de apoyarte en ella. Si voy a hacer scaffolding desde una plantilla, y especialmente si un agente lo va a hacer, quiero saber que la plantilla se sostiene antes de que estampe cien proyectos con la misma cosa rota incorporada.

Verifica más que "¿se analiza?". Lee el manifiesto `template.toml` de la plantilla, confirma que el nombre y la descripción no están vacíos, y verifica que cualquier herramienta que la plantilla diga que requiere esté realmente en tu PATH. Luego recorre cada archivo y cada nombre de archivo en la plantilla y ejecuta el procesamiento de plantillas sobre ellos, así un placeholder incorrecto (un error de sintaxis, o una variable que el manifiesto nunca define) se detecta como error antes de que alguna vez hagas scaffolding desde ella. También marca una plantilla sin archivos, y advierte si el manifiesto no está configurado para ser excluido de la salida. Así que es estructural, pero va más allá de "los archivos están ahí": realmente ejerce el procesamiento de plantillas de la manera en que `init` lo hará, y te dice dónde se rompe.

Esa es la razón real por la que el scaffolding pertenece a esta CLI y no a alguna herramienta generadora separada a un lado. La misma razón que todo lo demás en fledge: es la misma superficie, para los mismos dos tipos de usuarios. Una persona ejecuta `fledge templates init` y se salta la hora aburrida. Un agente ejecuta exactamente el mismo comando, sin interacción, y levanta un proyecto nuevo desde el paso uno sin que un humano haga clic en un asistente. Desde el primer capítulo, detectar, introspeccionar, quizás hacer scaffolding, y luego ejecutar, el paso de scaffolding es el "quizás". Cuando el repositorio ya existe, fledge lo detecta. Cuando aún no existe, el primer movimiento real es levantarlo desde una plantilla, y luego todo lo demás, el verbo de build, el verbo de test, el verbo de review, funciona en él de inmediato, porque salió de la plantilla ya cableado de la manera en que fledge espera.

Un proyecto que ya habla fledge desde el inicio es lo que realmente buscaba. No solo "ahórrame el copiar-y-pegar", aunque lo hace. Un proyecto con scaffolding se construye, se prueba, y está listo para los mismos verbos que cualquier otro repositorio, desde el primer commit. El boilerplate que solía pegar a mano era, la mitad del tiempo, exactamente el cableado que hacía que un proyecto fuera

consistente con los demás. Incorporar el scaffolding a la CLI del ciclo de vida significa que esa consistencia es el valor predeterminado con el que nace un proyecto, no algo que añado después.

Así que sí, un proyecto nuevo comienza con el scaffold. E incluso cuando no recurro a `fledge templates init` por nombre, el proyecto obtiene la misma configuración cableada desde el primer commit de todas formas: spec-sync, fledge, augur, attest. El "init" real no es la plantilla, es el stack que entra. El comando es solo la forma rápida de llegar ahí. De cualquier manera, un proyecto mío nace ya con las herramientas que tienen todos los demás.

Revisión con IA en el ciclo

El tercer pilar es el que sorprende encontrar en un ejecutor de tareas. Ejecución de tareas, scaffolding, claro, esas son obviamente cosas del ciclo de vida. Pero ¿revisión de código con IA? ¿En la misma CLI con la que construyes y pruebas? Eso parece que pertenece a otro sitio, en su propia herramienta, en CI, en un bot en tus pull requests.

Pero no es así, y la razón es simple: la revisión es parte del ciclo, igual que construir y probar. Escribes algo, lo verificas, lo arreglas, vuelves a empezar. La revisión es el paso de verificación. Y si los agentes son los que están escribiendo el código ahora, que para mí en su mayoría lo son, entonces calificar ese código es solo otro verbo. Se sienta justo al lado de `build` y `test` porque hace el mismo trabajo que ellos: te dice si la cosa realmente es buena antes de seguir adelante.

Una superficie le gana a tres herramientas, y le gana más a tres para un agente que para mí. Ese es el argumento completo para integrar la revisión en la CLI del ciclo de vida en lugar de publicar una herramienta separada. Un agente en una herramienta de revisión separada tiene que aprender toda otra cosa, con su propia invocación y su propia forma de salida, para hacer un ciclo continuo de trabajo. Si el agente ya sabe cómo manejar fledge para construir y probar, entonces `fledge review` es un verbo cuya forma ya conoce. Misma superficie, mismo JSON, mismo modo sin cabeza. No hay que aprender ninguna herramienta nueva solo porque el paso cambió de "¿compila?" a "¿es bueno?".

En fledge esto es `fledge review`, y lo que hace es revisión de código con IA contra la rama predeterminada. Así que no está revisando el mundo entero. Está mirando lo que cambió, tu diff contra la rama en la que te fusionarías, que es exactamente la unidad en la que ocurre la revisión. El mismo alcance que miraría un revisor humano o un bot de PR, ejecutado como un verbo en la misma CLI en la que ya vives.

Y no está atado a un modelo o proveedor. Bajo el capó, la revisión pasa por el mismo cliente multiproveedores que usa el resto de mis herramientas, corvid-ai, así que fledge habla con cualquier proveedor que corvid-ai soporte, la API de Anthropic, cualquier endpoint compatible con OpenAI, y ejecutores locales como Ollama, entre otros. La lista de proveedores compatibles vive en el repositorio de corvid-ai y cambia a medida que lo hace el ecosistema. El punto es que la revisión se ejecuta contra cualquier modelo que quieras, tu elección, no la de la herramienta.

Vale la pena ser directo sobre lo que eso significa, porque el resto de este libro es directo sobre el radio de explosión: `fledge review` toma tu diff y tus especificaciones, y los envía al proveedor al que lo apuntaste. Si ese es un endpoint alojado, tu código sin fusionar acaba de salir del edificio. Para un repositorio privado esa es una superficie real de egreso de datos, y es tu decisión tomada con los ojos abiertos, no un detalle a pasar por alto. Apuntarlo a un modelo local es la versión donde nada sale de la máquina. Un matiz honesto sobre ese caso local: la ruta de Ollama acepta felizmente una clave y una URL en la nube, pero el README de corvid-ai todavía enmarca Ollama como la opción local sin clave, así que los docs parecen que Ollama significa la caja bajo tu escritorio. La ruta en la nube funciona hoy; el README simplemente no ha llegado a eso. Esa es una brecha de documentación de mi parte, no una característica faltante.

También hay un ángulo de múltiples modelos que me gusta mucho. `--with-model` te permite ejecutar un panel, críticas paralelas sobre el mismo diff desde más de un modelo a la vez.

```
fledge review --with-model ollama:gpt-oss:120b-cloud,ollama:qwen3-coder:480b-cloud
```

Eso no es un truco. Si has pasado tiempo con estos modelos sabes que no todos detectan las mismas cosas, y tampoco alucinan las mismas cosas. Ejecutar un par de ellos sobre el mismo diff y ver dónde están de acuerdo, y dónde uno marca algo que los demás pasaron por alto, es una señal mejor que confiar en cualquiera de ellos por sí solo. Es una segunda y tercera opinión, ejecutadas en paralelo, sobre el cambio exacto que tienes delante.

Y como todo lo demás en fledge, la salida es estructurada. Cada comando emite `{schema_version: 1, ...}`. JSON por defecto. Así que una revisión no es un muro de prosa que un agente tiene que leer e interpretar como lo haría una persona. Son datos. El agente que escribió el código puede ejecutar la revisión, recibir los hallazgos estructurados de vuelta, y actuar sobre ellos en el mismo ciclo, sin un humano en el medio traduciendo "el revisor parece insatisfecho con el manejo de errores" en algo que realmente hacer. Una persona puede leer la revisión, y un agente puede analizarla, del mismo comando.

`fledge review` también es consciente de las especificaciones, que el capítulo de spec-sync cubre en detalle; aquí solo vale ver qué hace la revisión con una especificación cuando la tiene. La revisión descubre qué especificaciones cubren los archivos en el diff, haciendo coincidir en los archivos declarados de la especificación y en el directorio `specs/<nombre>/`, e incorpora esas especificaciones en el prompt como

contexto. Y apunta hacia ellas deliberadamente: se le dice al modelo que las especificaciones describen lo que los módulos *deben* hacer, que las use para interpretar el cambio, que revise solo el diff y no las especificaciones en sí mismas, y, la parte que importa, si el diff contradice un invariante de la especificación, que lo señale como un error en el diff. Así que la deriva de la especificación no es un sabor de fondo en la crítica; es un hallazgo que la revisión está explícitamente instruida a mostrar.

Y se gana su lugar. `fledge review` me ha detectado un error real. Algo que habría llegado a publicarse, ahí mismo en un diff que compilaba bien y pasaba sus pruebas, que el LLM marcó antes de que se fusionara. Esa es la prueba de si la revisión-como-verbo vale algo, y la pasó: no una observación de estilo, un defecto real que el resto del ciclo había dejado pasar. El lado consciente de especificaciones también ha dado resultados. Apuntar al revisor hacia las especificaciones ha detectado código que silenciosamente se alejó del contrato que se suponía que el módulo debía honrar, una deriva que compila y pasa igual que lo hizo el error. Los agentes se apoyan bien en ello, porque para ellos tiene la misma forma que build y test: ejecútalo, lee los hallazgos estructurados, arregla lo que encontró, vuelve a empezar.

Si los agentes escriben el código, ejecutan el build y ejecutan las pruebas, calificar el código es un verbo más al alcance que ya conocen, y detecta cosas que los demás dejaron pasar.

Plugins, y no confiar en ellos

fledge tiene un núcleo pequeño que hace poco, con toda la capacidad real en los plugins. Eso es a propósito. Si el núcleo tuviera que saber sobre cada lenguaje, cada flujo de trabajo, el paso raro de cada equipo, se deterioraría. Así que no lo hace. El núcleo conoce los verbos comunes y cómo detectar un proyecto; el alcance real viene de los plugins que se añaden alrededor, y cualquiera puede agregar uno sin tocar el núcleo.

La otra razón para llevar la capacidad hacia los plugins es que no quería ser dueño de la lista de lo que fledge puede hacer. La gente lo extiende como quiere: Rust, Swift, TS, shell. Escribes el plugin en lo que te sientas cómodo. No estás obligado a mi lenguaje para extender mi herramienta.

La manera en que un plugin habla con el núcleo es un contrato real y versionado: un binario con un manifiesto `plugin.toml`, hablando JSON con fledge, igual ya sea nativo o un módulo WASM en sandbox. El capítulo del protocolo tiene el formato de wire y el handshake de capacidades; aquí basta saber que la costura es un contrato, no un montón de convenciones.

Y para evitar una pregunta que surge: los plugins y los lanes no son lo mismo. Un plugin agrega una capacidad, un nuevo verbo. Un lane encadena verbos que ya tienes en un pipeline ordenado. Así que no es "todo es un plugin". Hay un núcleo real con los tres pilares integrados, los plugins extienden las capacidades a su alrededor, y los lanes concatenan esas capacidades en secuencias. Dejo los lanes en su propio capítulo más adelante.

Cualquier lenguaje, lo que significa que no puedes confiar en ellos

La otra cara de "extiéndelo como quieras, en cualquier lenguaje" es que acabas ejecutando código que no escribiste y no puedes garantizar. Un plugin es solo el programa de alguien más. Una vez que has decidido que cualquiera puede escribir uno en cualquier cosa, también has decidido que vas a ejecutar mucho código no confiable.

Así que fledge pone los plugins en sandbox con WASM, en Wasmtime. El objetivo es la seguridad. Un plugin no puede leer todo tu disco ni llamar a casa a menos que lo

permitas. Por defecto está encajonado: obtiene lo que le concedes y nada más.

Hay una segunda razón para el sandbox, y es la real. Los agentes ejecutan estos plugins. Si los agentes están instalando y ejecutando plugins, no puedes confiar en ellos por defecto. No en los plugins, y no, a ciegas, en la decisión de ejecutarlos. Un agente que alcanza, toma un plugin, y lo ejecuta es exactamente la situación donde "probablemente está bien" no es suficiente. El sandbox hace que las herramientas ejecutadas por agentes sean seguras. Ese es el puente hacia el material de confianza y radio de explosión en el que me adentro en el libro de agentes; aquí la versión del lado de las herramientas es simple: código no confiable más una cosa automatizada ejecutándolo equivale a que necesitas una caja alrededor de él. WASM/Wasmtime es la caja.

La elección de WASM sobre las alternativas obvias vale nombrarse. Las dos a las que recurrirías primero son los perfiles de seccomp (filtrado de syscalls de Linux) y los contenedores (Docker o similar). Ambos funcionan, pero ambos añaden fricción o suposiciones que no encajan aquí. seccomp requiere privilegio a nivel de SO para configurarse y es solo para Linux, así que rompe la promesa multiplataforma de inmediato. Los contenedores significan un demonio de Docker, una extracción de imagen separada, y un límite de proceso más pesado que "ejecutar un plugin". WASM encaja de manera diferente: Wasmtime se incrusta directamente en el proceso de fledge como una biblioteca, se incluye dentro del binario único, se ejecuta en cualquier SO sin demonios adicionales ni privilegios, y aplica el límite de capacidad estructuralmente en tiempo de enlace en lugar de a través de una política del kernel que alguien tiene que configurar. No es que seccomp o los contenedores estén mal. Es que una herramienta que se distribuye como un binario para ejecutarse en cualquier lugar necesita un sandbox que viaje con ella, y Wasmtime lo hace.

El canario

Un sandbox que no puedes probar es solo una esperanza. Así que hay un canario, un plugin cuyo trabajo es intentar hacer las cosas que un plugin en sandbox no debe poder hacer, y confirmar que no puede. Es la prueba de que el sandbox aguanta. Si el canario no puede escapar, el límite es real; si alguna vez pudiera, lo sabrías de inmediato.

En realidad son dos plugins. `fledge-plugin-canary` es el nativo. Corre sin sandbox y prueba que los ataques *funcionan*: leer claves SSH y credenciales de AWS, extraer variables de entorno como `GITHUB_TOKEN`, abrir conexiones de red, generar procesos, escribir en `.git/hooks`. Luego `fledge-plugin-canary-wasm` ejecuta la misma batería dentro del sandbox de Wasmtime, donde cada uno de esos debería volver como

BLOQUEADO. Su propia descripción es directa al respecto: "prueba que el sandbox de Wasmtime bloquea cada ataque que expone el canario nativo."

El punto es más agudo cuando ves los dos uno al lado del otro en lugar de tomar mi palabra por ello. La propia salida del canario nativo se contrasta a sí mismo con lo que vería un plugin WASM ejecutando el mismo código:

```
NATIVE: ~/.ssh/ READABLE            → WASM: BLOCKED (no preopened dir for ~)
NATIVE: ~/.config/fledge/ READABLE   → WASM: BLOCKED (outside sandbox)
NATIVE: GITHUB_TOKEN LEAKED          → WASM: BLOCKED (not passed to guest)
```

El mismo ataque, dos límites. El nativo lee tu clave SSH; WASM no puede, porque no hay directorio preabierto a través del cual pueda alcanzar ~. El nativo hereda `GITHUB_TOKEN`; WASM no puede, porque el entorno del invitado está vacío. Cualquier resultado que vuelva como FILTRADO en lugar de BLOQUEADO en el lado WASM significa que el sandbox escapó. Los dos juntos son la verificación de extremo a extremo: uno muestra que el peligro es real, el otro muestra que la caja aguanta.

Lo que el sandbox no protege

El sandbox es contención del radio de explosión. Limita lo que el código de un plugin puede alcanzar. No limita todo, y venderlo como cura para problemas que no resuelve sería la versión deshonesta de este capítulo. Así que aquí está la parte que WASM no te da.

No detiene el egreso de datos más arriba en la pila. La caja de WASM evita que un plugin alcance tu clave SSH, pero no rige los datos que entregas a una herramienta a propósito. `fledge review` envía tu diff y tus especificaciones al proveedor de LLM al que lo apuntaste, y si ese es un endpoint alojado, el código sin fusionar acaba de salir de la máquina. Ningún límite de WASM toca eso, porque los datos salen por la puerta principal, no a través del plugin. Ese es un problema de consentimiento y política, y lo trato como tal en el capítulo de revisión, no como algo que el sandbox resuelve.

No protege contra un plugin que se ejecuta como el usuario anfitrión. No todos los plugins son WASM. Los plugins nativos se ejecutan con tus permisos, tu entorno, tu disco. Esa es la razón completa por la que el canario nativo puede leer tu clave SSH: no está en sandbox. Cuando ejecutas un plugin nativo, o uno que bifurca un proceso desde debajo del invitado, lo estás confiando de la manera en que confías en cualquier cosa que ejecutas en tu máquina. El sandbox es la garantía de la ruta WASM, no una general sobre cada plugin.

Y no valida el origen o la intención. WASM contiene lo que el código puede hacer. No dice nada sobre si el código, o el agente que escribió el código que ejecuta el plugin, debería estar haciéndolo. Un plugin ejecutando lógica escrita por agentes sigue ejecutando lógica que yo no revisé, dentro de la caja. El código no confiable en una caja sigue siendo código no confiable. La caja solo evita que el daño se propague.

Así que el encuadre honesto es estrecho: WASM/Wasmtime es contención del radio de explosión agnóstica al origen para la ruta de ejecución de código. No es control de flujo de datos, no es gestión de consentimiento, y no es razón para dejar de pensar en qué le entregas a un plugin o quién lo escribió.

El ecosistema de plugins es más grande de lo que habría esperado si solo conoces fledge por sus tres integrados nativos: muchos repositorios `fledge-plugin-*` en la organización CorvidLabs en varios lenguajes (el capítulo del ecosistema los cubre), escritos en cualquier lenguaje que se adaptara al trabajo. El sandbox es lo que permite que eso sea cierto: un plugin puede estar escrito en cualquier cosa y seguir siendo seguro de ejecutar. Hago el recorrido completo del ecosistema y los lenguajes en su propio capítulo.

Cómo encaja fledge en el día a día

Los agentes ejecutando plugins no confiables a toda velocidad en muchos repositorios es por eso que el sandbox no es negociable. La cosa que toma esas decisiones no suele ser yo decidiendo caso por caso. Son agentes, corriendo constantemente, en muchos repositorios. El sandbox de WASM por debajo es lo que hace seguro dejar que eso funcione.

El protocolo de plugins

Esta es la costura, el lugar real donde un plugin y el núcleo se encuentran. Tiene un nombre en el repositorio: el protocolo está versionado, y la cadena de versión es `fledge-v1`. Un plugin lo declara, y ese es el handshake. Todo lo demás cuelga de él.

Lo que realmente escribe un autor de plugin

Un plugin es un repositorio git con un manifiesto en la raíz llamado `plugin.toml`, más uno o más ejecutables. El manifiesto es corto. Nombras el plugin, le das una versión, dices qué protocolo hablas, y listas los comandos que estás añadiendo:

```
[plugin]
name = "fledge-deploy"
version = "0.1.0"
protocol = "fledge-v1"

[[commands]]
name = "deploy"
binary = "bin/fledge-deploy"

[capabilities]
exec = true
store = true
metadata = false
```

Ese es el contrato completo del lado del autor. Cada entrada `[[commands]]` es un nuevo verbo que fledge conocerá, apuntado a un binario para ejecutar. El bloque `[capabilities]` es el autor diciendo, de antemano, qué necesita poder hacer este plugin: ejecutar comandos de shell, persistir un poco de estado, leer metadatos del proyecto. El valor predeterminado es `false`. Pides lo que necesitas y nada más, y la petición es visible en el manifiesto, no enterrada en el código.

Cómo habla el anfitrión con un plugin

Cuando invocas un comando de plugin, fledge genera el binario y habla con él a través de stdin y stdout como líneas JSON, un objeto JSON por línea. Lo primero que envía el anfitrión es un mensaje `init`, y ese mensaje es la idea de cero configuración hecha literal. Le entrega al plugin la situación completa: el nombre y la raíz del

proyecto, el lenguaje detectado, el estado de git (rama, sucia o limpia, remota), la propia versión y directorio del plugin, la versión de fledge, y las capacidades exactas que se concedieron. El plugin no tiene que ir a descubrir dónde está o qué tiene delante. El anfitrión ya lo sabe, ese es el trabajo completo del núcleo, así que simplemente se lo dice al plugin.

Después de eso es una conversación. El plugin envía mensajes de vuelta: `prompt`, `confirm`, `select` para preguntar algo al usuario; `log`, `progress`, `output` para informar; `exec` para ejecutar un comando; `store` y `load` para su pequeño parche de estado. Cualquier cosa con un `id` recibe exactamente una respuesta de vuelta: una `response` o un `cancel`. Stderr nunca se captura, así que un autor de plugin siempre puede imprimir debug directamente al terminal.

Es lo suficientemente sencillo como para leerlo por encima del hombro de alguien. El anfitrión abre con `init`, entregando la situación:

```
{"type": "init", "protocol": "fledge-v1",
 "args": ["staging"],
 "project": {"name": "my-app", "root": "/Users/dev/my-app", "language": "rust",
   "git": {"branch": "main", "dirty": false, "remote": "origin"}},
 "capabilities": {"exec": true, "store": true, "metadata": false}}
```

El plugin, ya sabiendo dónde está, le pide a fledge que ejecute un comando:

```
{"type": "exec", "id": "6", "command": "git tag -l 'v*' --sort=-v:refname",
"timeout": 10}
```

Y el anfitrión responde al `id` coincidente:

```
{"type": "response", "id": "6", "value": {"code": 0, "stdout":
"v0.9.1\nv0.9.0\n", "stderr": ""}}
```

Esa es la forma completa: `init` entrega todo lo que un agente de otro modo tendría que adivinar como JSON estructurado en lugar de prosa en un README, y cada solicitud con un `id` recibe exactamente una respuesta coincidente. La herramienta le dice al plugin dónde está; el plugin no busca.

Dónde se atornilla el sandbox

Todo lo anterior describe un plugin nativo, un ejecutable normal, hablando JSON a través de tuberías. El problema, y el repositorio es directo al respecto, es que un plugin nativo se ejecuta como tú, con tu acceso completo. El bloque de capacidades

controla el protocolo, pero no controla el *proceso*. Un plugin que no pidió nada podría seguir leyendo tus claves SSH, porque es solo un programa ejecutándose como tu usuario. Esa es la fuga que el canario probó: declarar capacidades al nivel de protocolo era teatro de seguridad, y es la razón completa del movimiento a WASM. Conté ese arco en el capítulo del sandbox; aquí el punto es solo qué cambió en el protocolo.

Los plugins WASM mantienen exactamente el mismo protocolo `fledge-v1` pero cambian el límite. El plugin declara `runtime = "wasm"` y envía un único binario `.wasm`. En lugar de stdin y stdout, los mismos mensajes JSON cruzan a través de tres funciones anfitrionas (`recv`, `send`, `exit`) que el anfitrión enlaza. Los mismos tipos de mensajes, la misma conversación. Y las capacidades dejan de ser una petición cortés. Se aplican en tiempo de enlace: si no concediste `exec`, la importación de `exec` simplemente no está enlazada, y un plugin que intente llamarla falla al instanciarse del todo. No hay verificación en tiempo de ejecución que engañar, porque la función que llamaría no existe para él. El acceso al sistema de archivos es `none`, `project`, o `plugin`, montado como directorios preabiertos; la red es un booleano. Además de eso, el runtime está limitado en combustible y en memoria, así que un plugin no puede girar para siempre ni devorar la máquina.

Así que la capacidad que ves en `plugin.toml` y la capacidad a la que el código puede realmente llegar son la misma cosa, estructuralmente.

La razón por la que tenía que ser estructural es la lección que enseñó el canario: una capacidad que simplemente *declaras* es una capacidad en la que confías en que el plugin la respetará, y la razón completa por la que pones un plugin en sandbox es que no confías en él. La única versión que aguanta es aquella en la que la capacidad que no concediste no es alcanzable, donde la función literalmente no existe para llamar. Cuando los agentes son los que instalan y ejecutan estas cosas, eso es lo que quieres.

Vale la pena nombrar aquí un riesgo de versión. La cadena de protocolo que un plugin declara es `fledge-v1`. Cuando llegue un cambio de protocolo que rompa la compatibilidad, y llegará en algún momento, esa cadena se convierte en `fledge-v2`. Un plugin que todavía declara `fledge-v1` contra un anfitrión que espera `fledge-v2` falla en la instanciación, ruidosamente, antes de que cualquier código se ejecute. No se comporta mal silenciosamente. La versión en el manifiesto es lo que fuerza ese fallo temprano: un plugin no coincidente no llega al punto donde puede hacer nada, así que el operador sabe exactamente qué necesita actualizarse en lugar de perseguir un error de runtime sutil.

Hooks del ciclo de vida de lanes

El mismo protocolo lleva un segundo tipo de integración: hooks del ciclo de vida. Un plugin puede registrar un hook en su `plugin.toml` que se dispara en eventos `lane:pre` o `lane:post` en lugar de en una invocación directa de comando. El plugin de deploy es el ejemplo que se incluye en los docs: registra un hook `lane:post` que se ejecuta automáticamente después de que cualquier lane de fledge se completa.

Cuando se dispara el hook, fledge pasa contexto al binario del hook a través de variables de entorno: `FLEDGE_LANE_NAME`, `FLEDGE_LANE_STATUS`, y `FLEDGE_LANE_RUN_ID`. La definición del lane no tiene conocimiento de qué plugins están instalados. El plugin no tiene conocimiento de qué lanes existen. El evento del ciclo de vida es la costura.

Aquí está el aspecto completo. El lane vive en `fledge.toml`; el registro del hook vive en el `plugin.toml` del plugin; los dos se conectan sin que ninguno conozca los detalles del otro:

```
# fledge.toml
[lanes.verify]
description = "Pre-merge gate: format, lint, test, build"
steps = ["fmt", "lint", "test", "build"]
```

```
# plugin.toml (from fledge-deploy, or any plugin registering a lane hook)
[plugin]
name = "fledge-deploy"
version = "0.2.0"
protocol = "fledge-v1"

[[commands]]
name = "deploy"
binary = "bin/fledge-deploy"

[[hooks]]
event = "lane:post"
binary = "bin/fledge-deploy-hook"

[capabilities]
exec = true
store = true
metadata = false
```

Cuando `fledge lanes run verify` se completa, fledge dispara el evento `lane:post`. Cualquier plugin instalado con una entrada `[[hooks]]` coincidente recibe su binario de

hook llamado con `FLEDGE_LANE_NAME=verify`, `FLEDGE_LANE_STATUS=success` (o `failure`), y `FLEDGE_LANE_RUN_ID` configurado.

El lane emite contexto estructurado como variables de entorno en lugar de prosa en stdout, así que un plugin lee un estado sobre el que puede actuar en lugar de raspar texto y adivinar. Ese es el mismo principio de diseño que el mensaje `init`: el anfitrión te dice qué pasó; no tienes que inferirlo.

Los tres pilares se sientan encima de esto

Los tres pilares (ejecución de tareas, scaffolding, revisión con IA) son núcleo, no plugins de la comunidad. El protocolo es la capa de extensión *alrededor* de ellos. La columna vertebral es de fledge, y `fledge-v1` es cómo cualquiera atornilla más capacidad en él. El núcleo es real y los plugins lo rodean; el protocolo es solo la costura donde los dos se encuentran.

El ecosistema de plugins

Hay docenas de repositorios `fledge-plugin-*` en la organización CorvidLabs, en varios lenguajes, con algunos archivados. El conteo en vivo es real, pero es lo menos interesante del ecosistema. El conteo de plugins es inventario. El argumento es cuáles te daría realmente, y qué tan seguro estoy de cada uno.

Así que aquí están los cinco que usarías a diario: `augur`, `attest`, `format`, `gitleaks`, `deps`. Empieza por ahí. El resto de este capítulo trata sobre por qué existen los otros treinta y siete y cuánto confiar en ellos.

Los cinco a los que recurrirías

Estos corren en básicamente cada cambio, el mío y el de los agentes. Se agrupan en dos trabajos.

El par de confianza es `augur` y `attest`. `augur` puntúa el riesgo de cambio de un diff, proceder, revisar o bloquear, a partir de señales estructurales solas, sin clave de API y sin LLM, y su descripción dice en voz alta a quién va dirigido: "para humanos y agentes." `attest` registra la procedencia firmada de quién revisó qué commit, guardada en git notes. Esos dos son los que más me apoyo, porque son los que importan cuando un agente es el autor. Puntuación de riesgo a la entrada, un rastro de procedencia a la salida.

Los tres del ciclo de desarrollo son aburridos a propósito: `format`, `gitleaks`, `deps`. Ejecuta el formateador. Escanea en busca de secretos comprometidos. Verifica el estado de las dependencias, desactualizadas, auditoría, licencias, en Rust, Node y Python. Ninguno de ellos es emocionante. Todos ellos se ejecutan constantemente, porque ese es exactamente el tipo de verificación que quieres que se dispare automáticamente en lugar de que tengas que recordarla.

Si solo instalaras esos cinco, tendrías la mayor parte de lo que obtengo del ecosistema.

Tres niveles, y qué tan seguro estoy

La manera honesta de leer el roster es por confianza, no por conteo. Los plugins caen en tres niveles, y confío en ellos de manera muy diferente. Consulta el roster en vivo

en la organización CorvidLabs para la lista actual; lo que sigue es cómo leer lo que encuentres allí.

Núcleo. `augur`, `attest`, `format`, `deps`, `gitleaks`. Los cinco diarios. Los escribí yo, los ejecuto en cada cambio, y están mantenidos y probados. Este es el nivel en el que apostaría el flujo de trabajo.

Integración. `github`, `discord`, `algochat`, `memory`. Estos conectan fledge con algo externo: la API de GitHub a través de `gh`, webhooks de Discord para notificaciones de fallos de CI, mensajería cifrada en cadena, un almacén de memoria. También están mantenidos y probados, pero llevan el riesgo de lo que sea que hablan. Una integración es solo tan confiable como el servicio detrás de ella.

Campos de prueba. `weather`, `roast`, `hangman`, y el resto de los de uso único. `weather` imprime un pronóstico en el terminal. `hangman` juega al ahorcado con identificadores extraídos de tu código base. `roast` pasa un commit por el LLM y lo critica, "solo para entretenimiento." Estos no están mantenidos al mismo nivel. Existen para demostrar que el sandbox funciona: si un plugin de clima medio-serio escrito a capricho es seguro de ejecutar, el protocolo está haciendo su trabajo. El bajo nivel no es una debilidad. Es la prueba.

Núcleo e integración están mantenidos y probados. Los campos de prueba demuestran que el sandbox aguanta. No leas el conteo plano como una distribución uniforme de herramientas igualmente verificadas, porque no lo son, y pretender lo contrario sería la versión deshonesta de este capítulo.

Procedencia, claramente

La mayoría de estos los escribí yo. Los cinco diarios son míos. También lo son casi todos los de uso único, escritos a capricho porque publicar un plugin no cuesta nada una vez que el protocolo existe. Necesitaba algo, escribí un plugin, lo añadí. El punto del protocolo es que no tengo que hacer crecer el núcleo para añadir una capacidad.

No voy a pretender que tengo una comunidad de autores externos que no tengo. Algunos plugins vienen del círculo, pero no presento el roster como amplitud verificada por la comunidad. Es principalmente una persona llenando un cajón, y esa es la procedencia honesta.

El canario, y por qué todo esto es seguro

Un par se gana su propia mención porque es fundamental para todo el ecosistema: `fledge-plugin-canary` y `fledge-plugin-canary-wasm`, el par de red team del capítulo del

sandbox. El nativo demuestra que los ataques funcionan. El WASM demuestra que el sandbox los bloquea. Todo lo demás aquí, un plugin escrito en Kotlin, un plugin que escribí medio dormido, solo es seguro de ejecutar porque ese par mantiene el límite. El ecosistema puede ser caótico precisamente porque la caja a su alrededor no lo es.

Por qué tantos lenguajes

La mayoría son Rust y shell, luego Swift, TypeScript, Kotlin, Python, y algunos de HTML/JavaScript. Esa distribución es el protocolo funcionando según lo planeado. La razón completa de `fledge-v1` y el sandbox de WASM es que un plugin puede estar escrito en cualquier cosa y seguir siendo seguro de ejecutar, así que el ecosistema no tiene que ponerse de acuerdo en un lenguaje, solo en un contrato. `fledge-plugin-bridge` es Kotlin. `fledge-plugin-memory` es TypeScript. `fledge-plugin-attest` es Swift. La mitad de las integraciones son shell. Nadie tuvo que aprender Rust para añadir a fledge. Escribieron la cosa en lo que tenían delante, y el núcleo se mantuvo del mismo tamaño todo el tiempo.

No soy dueño de la lista de lo que fledge puede hacer. El protocolo lo es, y le permite a cualquiera añadir a ella sin preguntarme. La distribución es la prueba, y los niveles son la honestidad al respecto.

---

spec-sync y mantener la honestidad

La deriva de especificaciones es el modo de fallo que me importa más que la mayoría. Escribes una especificación, el contrato para un módulo, qué hace, cuál es su API pública, y luego el código se desvía. Alguien cambia una función, la especificación todavía afirma la forma anterior, y ahora el documento y el código discrepan en silencio. Con un equipo humano eso es un README obsoleto. Con un agente en el ciclo es peor, porque el agente lee la especificación como verdad, construye sobre un contrato que el código ya no honra, y nadie se da cuenta hasta que algo se rompe. spec-sync existe para hacer que esa deriva sea imposible de ignorar.

spec-sync es su propia herramienta, su propio binario en Rust, su propia GitHub Action, y su tagline dice lo que es: "Validación bidireccional especificación-código con referencias entre proyectos, grafos de dependencias y generación potenciada por IA." Las dos palabras que importan son *bidireccional* y *validación*. Verifica, en ambas direcciones, que la especificación y el código estén de acuerdo. No es una suite de pruebas y no es un diff difuso contra prosa. Es verificación estructural de contratos: si la API pública documentada realmente coincide con la real. Las dos direcciones no son simétricas: una exportación que el código tiene pero la especificación no documenta es una advertencia, algo que completar; una entrada que la especificación afirma que el código no tiene es un error, una promesa rota.

Qué verifica realmente

Una especificación es un archivo markdown, `*.spec.md`, con frontmatter YAML y un conjunto de secciones requeridas. El frontmatter nombra el módulo, una versión, un estado, y los archivos fuente que cubre. Las secciones `##` requeridas son Purpose, Public API, Invariants, Behavioral Examples, Error Cases, Dependencies, y Change Log. Si falta una sección requerida, la especificación está incompleta y la validación bloquea.

Esa es la regla; aquí hay una especificación que la honra. Esta es la cabeza de la propia especificación del módulo `review` de fledge, el frontmatter real, y la sección `## Public API` completada con las exportaciones reales que spec-sync verifica contra el código:

```
---
module: review
version: 10
status: active
files:
  - src/review.rs
db_tables: []
depends_on:
  - spec
  - llm
  - config
---

# Review

## Purpose

AI-powered code review of current branch changes...

## Public API

### Exported Functions

| Export | Description |
|--------|-------------|
| `run` | Entry point for the review command |
| `ReviewOptions` | Options struct: base, file, json, model, provider,
with_model |
| `ReviewFormat` | Enum: Summary, Checklist, or Inline |
```

La línea `files:` es lo que spec-sync intersecta contra el diff y contra las exportaciones reales de `src/review.rs`. Cada fila en esa tabla `## Public API` es un nombre que spec-sync espera encontrar en el código; una entrada que el código no tiene es el caso de error a continuación. Si nunca has escrito una especificación, ese frontmatter más una tabla `## Public API` honesta es la forma a copiar. El resto de las secciones requeridas son del mismo tipo, prosa y tablas que describen lo que el módulo promete.

Luego valida en ambas direcciones:

- **Código a especificación:** exportación no documentada, advertencia.
- **Especificación a código:** entrada fantasma o obsoleta, archivo fuente faltante, incompatibilidad de tipos, todos errores.

Hace lo mismo para los esquemas: las tablas y columnas de bases de datos declaradas se verifican contra el SQL real, y una tabla o columna fantasma falla. Y resuelve referencias entre proyectos, así que una especificación en un repositorio puede depender de un módulo en otro a través de `propietario/repo@módulo` y ese enlace también se verifica.

Lo que hay que retener es que esto es *estructural*. No está calificando tu prosa y no está generando pruebas. Está haciendo una pregunta directa: ¿la superficie documentada coincide con la superficie real? Y responde de manera determinista. Eso es lo que hace que sea seguro ponerlo delante de un agente. No hay juicio de valor con el que discutir; o la API coincide con la especificación o no.

Cómo aparece

Aparece de tres maneras, y en la entrevista la respuesta a "¿cómo aparece spec-sync en el día a día?" fue: todas ellas. Tres puertas de entrada al mismo formato de especificación, cada una ejecutándose en un lugar diferente:

Puerta de entrada	Dónde se ejecuta	Qué hace
`fledge spec` (nativo)	tu máquina, dentro de fledge	`init`, `check`, `list`, `show`: la propia validación de fledge, sin shell-out
binario `specsync`	el ciclo del agente, antes de un PR	`specsync check` / `--fix`: la herramienta independiente que el agente ejecuta en el ciclo
`CorvidLabs/spec-sync@v4`	CI, en el pull request	bloquea el build, comenta la deriva, bloquea en caso de fallo

El resto de esta sección son esas tres filas, una a la vez.

fledge conoce las especificaciones de forma nativa. spec es uno de los pilares de fledge. El README de fledge es verbatim al respecto: "Spec | `spec` | [spec-sync]. Los módulos declaran su contrato, la IA lo usa como contexto." Vale la pena ser preciso sobre lo que eso significa en el código: fledge tiene su propio `fledge spec` (`init`, `check`, `list`, `show`) integrado directamente en el binario. No hace shell-out a la herramienta `specsync`. Lee el mismo `.specsync/config.toml` y los mismos archivos `*.spec.md`, los analiza por sí mismo, y hace su propia verificación. Así que el formato de especificación es compartido, pero la validación dentro de fledge es el propio código de fledge, no una llamada al binario separado.

Esa conciencia nativa de especificaciones es también por qué los comandos de IA de fledge pueden apoyarse en el contrato. `fledge ask` es preguntas y respuestas conscientes de especificaciones y `fledge review` es revisión de código consciente de

especificaciones; ambos extraen las especificaciones relevantes como contexto. La especificación es el contexto que esos comandos le alimentan al modelo. La especificación no es un documento a un lado; es la cosa que las herramientas leen cuando razonan sobre tu código.

Bloquea CI. Hay una GitHub Action independiente, `CorvidLabs/spec-sync@v4`, que ejecuta `specsync check` automáticamente. Un flujo de trabajo mínimo:

```
- uses: CorvidLabs/spec-sync@v4
  with:
    strict: 'true'          # warnings become errors
    require-coverage: '100' # minimum spec coverage
```

`strict` convierte las advertencias en errores. `require-coverage` establece un piso. `comment` publica un resumen de la deriva de especificaciones en el pull request. Y la verificación sale con código no cero en caso de fallo, lo que significa que el PR está bloqueado. Ese es el mecanismo que hace que todo sea real: la deriva no genera una nota cortés, falla el build. Un agente, o un humano, no puede dejar que la especificación y el código se separen en silencio, porque la separación *es* la verificación que falla.

Mantiene al agente dentro de la especificación. Este es el que más me importa, y es por qué la especificación vive dentro del ciclo del agente, no solo en CI. La mecánica es concreta. El agente ejecuta `specsync check` como parte de su ciclo, de la misma manera que ejecuta build y test. La verificación regresa con fallos estructurados, y el agente los lee y decide qué hacer, y hacia dónde va depende de en qué dirección corra la deriva.

Aquí está el aspecto de esa salida estructurada cuando ambas direcciones se han desviado (ejemplo ilustrativo que coincide con el formato de salida real):

```
specsync check

CHECKING src/review.rs against review.spec.md
  WARNING  undocumented export: `ReviewOptions::with_model`
             → run `specsync check --fix` to add stub

  ERROR    phantom export: `ReviewFormat::Detailed`
             spec claims this variant; code has: Summary, Checklist, Inline
             → update spec or add the variant to code

  ERROR    phantom export: `run_async`
             spec claims this function; not found in src/review.rs
             → update spec or add the function

SUMMARY  2 errors, 1 warning
EXIT 1
```

Eso es lo que lee el agente. Los errores nombran el archivo, la afirmación de la especificación, y lo que el código realmente tiene. Las advertencias nombran la exportación y el comando de corrección exacto. No hay que adivinar: o reconcilia el código para que coincida con la especificación o corrige la especificación, luego vuelve a ejecutar hasta exit 0.

Si el código creció una exportación que la especificación no menciona, la dirección de advertencia, el agente no edita la especificación a mano. Ejecuta `specsync check --fix`, que añade automáticamente las exportaciones no documentadas a la especificación como stubs, y el caso fácil se reconcilia en un movimiento. El trabajo del agente ahí es principalmente completar el stub con una descripción real, no descubrir la deriva en primer lugar.

La otra dirección es la que requiere trabajo real. Cuando la especificación afirma una API que el código no honra, la dirección de error, la entrada obsoleta o fantasma, no hay `--fix` para ello, porque la corrección es un juicio de valor: o el código está mal y el agente va a hacer que el código coincida con lo que se prometió, o la especificación era aspiracional y el agente corrige el contrato. De cualquier manera el agente tiene que reconciliarlo en el código, deliberadamente, y volver a ejecutar la verificación hasta que esté en verde. Ese es el ciclo en la práctica: contrato, cambio, verificación, y luego ya sea una adición automática o una corrección real dependiendo de qué lado se desajustó, y corre antes de que el diff llegue a un pull request, dentro del ciclo de Merlin, no después en CI.

Lo que spec-sync no verifica

Hay una línea sobre la que quiero ser honesto, porque es el límite de lo que hace esta herramienta. spec-sync verifica que la spec y el código estén de acuerdo. No tiene opinión sobre si la spec es buena.

Piensa en lo que eso deja abierto. Una spec puede nombrar las exportaciones correctas y describirlas mal. Puede ser delgada, una sección Purpose que no dice nada y una tabla Public API que es precisa y no te dice nada sobre para qué sirve el módulo. Puede ser aspiracional, escrita para el código que alguien tenía intención de escribir. Y si un agente redactó la spec a partir de un brief de tarea que era en sí mismo incorrecto, o inyectado, la spec puede ser un contrato fiel para la cosa equivocada. En cada uno de esos casos spec-sync pasa. La superficie coincide con la superficie. La verificación sale en verde. El contrato está mal, y nada en el ciclo lo capturó, porque verificar la spec contra el código no puede decirte que la spec era mala desde el principio.

Esa es una brecha real y la nombro como tal. El código tiene una verificación; la spec no. Lo que querrías es una segunda barrera que corre sobre la spec misma, antes de que un agente construya contra ella: ¿cada sección requerida realmente dice algo, apunta la Public API a archivos que existen, hay un enunciado claro de cómo se ven el éxito y el fallo, y firmó realmente un humano este contrato. Eso es un `fledge spec lint`, y todavía no está construido. Hasta que lo esté, la spec es la única entrada a todo este pipeline que nada valida, y vale la pena saberlo cuando confíes en la verificación en verde.

Por qué un contrato compartido es el objetivo completo

Esto se conecta directamente con la tesis bajo todas estas herramientas: humanos y agentes usando las mismas herramientas, como ciudadanos de primera clase iguales. Una especificación es la versión más clara de esa idea que tengo. El humano la escribe o la revisa; el agente construye sobre ella; spec-sync los mantiene a ambos a ella, en ambas direcciones, de manera determinista. El humano que refactorizó sin actualizar el documento queda atrapado de la misma manera que el agente que alucinó una API. La verificación no le importa cuál de los dos se desvió.

Construyendo fledge en Rust

fledge es un único binario. Lo instalas una vez y corre en cualquier máquina, cualquier SO, sin nada más que instalar junto a él. Sin runtime que traer, sin intérprete, sin gestor de dependencias que ya tenga que estar presente. Multiplataforma por defecto: un pipeline de build produce un binario que funciona en macOS, Linux y Windows. Ese es el requisito central, y Rust es la razón por la que es fácil en lugar de una pelea. Todo el capítulo se desprende de eso: por qué Rust fue la elección correcta, por qué aprenderlo no fue la historia de guerra que la gente espera, y cómo los agentes cubrieron la brecha de fluidez.

Había estado escribiendo Swift durante aproximadamente una década cuando empecé fledge. Así que la versión honesta de este capítulo es un poco anticlimática: aprender Rust no dolió. Nada importante me peleó. Ni siquiera el borrow checker, que es la parte ante la que todo el mundo te advierte.

Creo que la gente espera una historia de guerra aquí: el lenguaje que me humilló, el mes que pasé perdiendo peleas con el compilador. No la tengo. Y prefiero contarte por qué fue fluido que inventar el drama.

Swift me preparó

Mucho de Rust se sentía familiar porque Swift ya me había entrenado en las mismas ideas, solo con ropa diferente.

Los enums y la coincidencia de patrones fueron lo grande. Los enums de Swift son verdaderos tipos suma, y había estado apoyándome en ellos y en `switch` durante años. El `enum` y el `match` de Rust son el mismo músculo. Result y Option tampoco eran nuevos. Había vivido en los opcionales de Swift y `Result` durante mucho tiempo, así que "esto puede ser un valor o nada" y "esto puede ser un valor o un error" ya era cómo pensaba sobre el código. Rust simplemente te hace manejar ambos, todo el tiempo, en voz alta. Esa no era una nueva disciplina para mí; era una disciplina que ya tenía, ahora aplicada.

Los traits llegaron como aproximadamente los protocolos de Swift. No idénticos, pero lo suficientemente cercanos como para que no estuviera aprendiendo un concepto extranjero. Estaba aprendiendo un dialecto de uno que conocía. "Define comportamiento, conforma tipos a él" tiene la misma forma en ambos.

Así que el lenguaje no se sintió como un muro. Se sintió como un lugar donde había vivido a medias antes. Los hábitos de valor-tipo y opcionales que Swift me inculcó son, creo, exactamente por qué el borrow checker nunca se convirtió en una pelea. Si ya piensas en términos de propiedad y "quién tiene este valor," el checker principalmente te está diciendo cosas que ya estabas intentando hacer. No era una nueva forma de pensar que tuviera que adquirir. Era un árbitro más estricto para un juego que ya sabía cómo jugar.

Y la forma de lo que construye es la simple que quería: fledge es un único binario en Rust, construido con Cargo. Un crate, un binario al final.

Los agentes hicieron el trabajo pesado

No voy a pretender que soy un programador fluido en Rust. No lo soy. Soy un programador fluido en Swift que puede leer y dirigir Rust bien, y eso es algo diferente.

Lo que cerró la brecha fueron los agentes. Me apoyé en ellos mucho para ser productivo en un lenguaje en el que tengo menos fluidez. Las partes familiares podía leerlas, razonar sobre ellas y dirigirlas por mi cuenta. Sabía lo que quería que el código *hiciera* y cómo se veía una buena estructura, porque esa parte transfiere entre lenguajes. Las partes donde Rust tiene sus propios modismos, su propia manera de decir las cosas, los agentes las llevaron.

Esta es una distinción que vale la pena hacer, vista desde el otro lado. Mi Swift está escrito a mano. Mi Rust está amplificado por agentes. En uno soy el autor en las teclas. En el otro soy el que sabe cómo se ve lo correcto, apuntando agentes hacia ello y verificando su trabajo.

Y honestamente, construir fledge en Rust de esta manera es una pequeña prueba de toda la tesis detrás de él. La herramienta está construida para humanos y agentes por igual. Fue construida *por* un humano y agentes por igual. Los agentes que manejan fledge hoy ayudaron a escribir fledge en primer lugar.

Las herramientas fueron un alivio

Aquí está la parte que no esperaba disfrutar tanto como lo hice: las herramientas de Rust se sintieron como un alivio.

Cargo simplemente funciona. Una herramienta para construir, probar, dependencias, todo, y es la misma en todas partes. El ecosistema de crates es profundo. Lo que sea que necesitara, usualmente había un crate sólido para ello, y añadirlo era una línea.

Y las builds de binario único son exactamente lo que fledge quería ser: necesitaba publicar un único binario ligero que corriera en todas partes, y Rust te da eso por defecto.

El contraste es con las herramientas de Swift una vez que sales de las plataformas de Apple. En Apple, la historia de Swift es genial. Fuera de ellas (Linux, multiplataforma, el tipo de objetivo "corre en cualquier lugar como un único binario" que fledge necesitaba) se complica. Esa es una razón real por la que fledge es Rust y no Swift, aunque Swift sea mi lenguaje de origen. Quería un binario ligero y único que pudiera soltar en cualquier máquina, que los agentes pudieran ejecutar en cualquier lugar, y la cadena de herramientas de Rust hizo que ese fuera el camino fácil en lugar de la pelea.

Así que esa es la verdad desde la perspectiva del constructor. Swift preparó el pensamiento, los agentes cubrieron la fluidez que no tengo, y las herramientas (Cargo, los crates, el binario único) fue la parte que realmente me alegró haber cambiado. Elegir el lenguaje fue la decisión fácil. El trabajo que realmente requirió reflexión fue todo lo demás: descifrar qué debía ser fledge.

Cómo lo uso, y quién lo usa

Déjame empezar con la parte honesta en lugar de guardarla para el final. fledge no está ampliamente utilizado. Es mío, es de mis agentes, es de mi círculo. Ese es el punto. La herramienta se construyó para resolver el problema real del constructor primero, y lo hace, todos los días. Una herramienta que resuelve el problema frente a ti le gana a una con un muro de logos y sin nada en juego.

Así que la pregunta que responde este capítulo no es "cuánta gente usa fledge". Es "quién, y por qué ese es el orden correcto." La respuesta no es "todos", y nunca se suponía que lo fuera.

Mis agentes lo manejan más de lo que yo lo hago

Lo que la gente entiende mal cuando se imagina cómo uso fledge es que se imaginan *a mí* usándolo. Eso ocurre, pero no es la forma principal en que fledge corre ahora. Mis agentes lo manejan, por un amplio margen. Cuando un agente está trabajando en un repositorio, fledge es la manera en que construye, prueba y ejecuta lo que acaba de cambiar: su superficie de ejecución, no solo la mía. Eso es exactamente lo que el primer capítulo dijo para lo que estaba diseñado. La mayoría de los días los agentes sacan más provecho de él que yo, y yo me dedico principalmente a dirigir.

Esa es la base de usuarios para la que construí. No una multitud de extraños. Yo más los agentes, en cada repositorio que tengo, todo el tiempo.

Quién más lo usa

fledge es de código abierto, está en el tap de Homebrew, cualquiera puede hacer `cargo install`. Nada de eso es una afirmación de adopción. Un tap es un canal de distribución, no un conteo de usuarios, y no voy a vestir uno como el otro.

Hoy la base de usuarios soy yo, mis agentes, y el círculo de CorvidLabs, las personas con las que realmente trabajo. No hay adopción externa amplia. No hay comunidad de extraños presentando issues. Es infraestructura personal y del círculo, y vale la pena decirlo claramente en lugar de insinuar una corriente que no existe.

Los colaboradores importan para el panorama, no solo como conteo de cabezas. Que fledge sea la superficie compartida en todo el círculo es parte del diseño. Cuando alguien en CorvidLabs toma uno de mis repositorios, no tiene que aprender el

dialecto privado de ese repositorio. Ya conoce los verbos, porque los verbos son los mismos en todas partes. La misma consistencia que me ahorra el reaprendizaje a mí aterriza a las personas con las que trabajo en una superficie que ya conocen.

Si deberías usarlo

Quizás todavía no. Honestamente, solo si trabajas como yo: muchos repositorios, agentes en el ciclo, una superficie consistente en todos ellos. Si ese es tu problema, fledge lo resuelve. Si no lo es, la herramienta es excesiva y prefiero decírtelo que vendértela.

Lo que no voy a afirmar es que la adopción limitada demuestra que el diseño funciona a escala. No lo hace. Demuestra que el diseño funciona para mí, que es una afirmación más pequeña y la única que puedo respaldar. fledge se gana su lugar dentro de mi propio mundo primero, todos los días, principalmente a través de agentes, con yo dirigiendo y un pequeño círculo en la misma superficie. Eso es una herramienta haciendo su trabajo. La amplitud puede llegar después o nunca. El trabajo ya se está haciendo.

Hacia dónde va fledge

La respuesta honesta a "¿hacia dónde va fledge a continuación?" es menos emocionante de lo que la gente espera, y creo que es una buena señal. Está principalmente maduro. La gran forma de él está construida. Lo que queda es principalmente mantenerlo y seguir usándolo. Hay un par de direcciones en las que lo haría crecer, pero no estoy sentado sobre un gran plan de reinvención, porque fledge no necesita ser reinventado. Necesita seguir siendo la cosa sobre la que todo lo demás se apoya.

Un ecosistema más grande

Los niveles de campo de prueba e integración están abiertos: cualquiera puede añadir a ellos sin preguntar. El nivel central no lo está. Esa asimetría es intencional y no va a cambiar.

La dirección con más espacio es la que el capítulo del ecosistema ya describió: más plugins. fledge se vuelve más capaz por el anillo alrededor del núcleo que crece, no por el núcleo que engorda: un conjunto más rico de plugins para que en cualquier repositorio en que se deposite fledge, ya haya algo que sepa cómo manejarlo. Más integraciones, más de los de uso único que cada uno resuelve una cosa real.

Ligado a eso está ampliar lo que fledge detecta y ejecuta de entrada, la promesa de cero configuración del primer capítulo, mantenida en más lugares. Cada lenguaje y plataforma que aún no conoce es un repositorio donde la promesa de mismos-verbos-simplemente-funcionan tiene una brecha, así que completar la detección y la cobertura de plataformas es la otra dirección obvia. No es glamoroso. La cobertura es simplemente lo que hace que una superficie universal sea realmente universal.

Y la cobertura que más importa ahora mismo no es el siguiente lenguaje. Es el sistema operativo. fledge tiene que correr de primera clase en los tres: Windows, Linux y macOS. Una herramienta que solo está realmente en casa en un SO no es una superficie universal, es una local. Así que el soporte multiplataforma es la próxima frontera real, más que añadir otro detector de lenguaje.

Qué significa mantener algo de lo que dependen los agentes

Podría listar características aquí y prometer una reinvención. Ese sería el marco equivocado para lo que fledge es ahora.

La pregunta que importa en 2026 no es qué características agrega fledge. El código es barato. Los agentes escriben la mayor parte. Lo que escasea es una herramienta en la que realmente puedas confiar para hacer tu trabajo. Un agente puede generar nueva funcionalidad más rápido de lo que yo puedo revisarla. Lo que no puede generar es una herramienta con años de uso intensivo detrás de ella, una interfaz estable, un comportamiento en el que un pipeline puede contar.

Una herramienta con años de uso intensivo y una interfaz estable en la que un pipeline puede contar es lo que es fledge, y es la cosa más difícil de construir. No más difícil técnicamente. Más difícil porque requiere decidir que la estabilidad es la salida, no un subproducto. Cada herramienta comienza su vida como una nueva característica. Muy pocas sobreviven lo suficiente como para convertirse en infraestructura. Las que lo hacen son aquellas en las que el mantenedor trató "nada se rompió" como el logro principal, no como un premio de consolación por un sprint lento.

Uso fledge en cada repositorio, todo el tiempo. Mis agentes lo manejan constantemente. Eso significa que los errores me encuentran a mí. El verbo que falta, la detección que adivinó mal, el hook del plugin que se disparó en el orden equivocado. Lo encuentro porque vivo en él, y lo arreglo porque vivo en él. El uso intensivo no es una actividad secundaria. Es todo el mecanismo de calidad. Mantener y usar son el mismo trabajo.

La introducción de este libro preguntó qué hace que una herramienta valga la pena como dependencia. La respuesta que señaló fue: una superficie limpia, sin pasos interactivos ocultos, un límite de plugin real, una especificación que se mantiene honesta a medida que el código cambia. Constrúyela así, y un agente puede manejarla desde el primer día, porque nunca fue construida para necesitar una persona para empezar.

fledge es eso. No por ninguna decisión de diseño individual sino por el resultado acumulado de cada vez que me negué a dejar que un atajo erosionara la interfaz. Los verbos son los mismos en cada repositorio. El JSON tiene la misma forma. El manifiesto de `fledge introspect` se mantiene actualizado. Nada se publica sin que yo lo haya ejecutado primero en mi propio trabajo.

Cuando los agentes son los que operan la mayoría de las herramientas, lo que gana confianza no es una larga lista de características. Es aparecer de la misma manera cada vez, durante el tiempo suficiente para que algo real se construya encima de ello. Esa es una ambición más silenciosa que una diapositiva de hoja de ruta, y es la que importa.

Así que el trabajo de aquí en adelante es el que siempre ha sido: mantenerlo sólido, seguir viviendo en él, mantener la superficie honesta. Seguiré encontrando los errores porque sigo corriéndolo, y seguiré arreglándolos. Ese es el plan completo.

Sobre el Autor

0xLeif (leif.algo) construye en abierto. Una década de pequeñas y componibles bibliotecas Swift como AppState, Cache y Fork. El laboratorio CorvidLabs. Una pila de herramientas de agentes que empezaron principalmente como "ojalá esto existiera". Fuera del teclado es Zach Eriksen.

Estos libros son entrevistas, moldeadas en capítulos y verificadas contra el código real.

github.com/0xLeif · leif.algo

Agradecimientos

Gracias a CorvidLabs, por ser la sala donde estas ideas se prueban y se discuten hasta tomar forma.

Gracias a los mantenedores de código abierto cuyas herramientas sostienen toda esta pila. Nada de esto se construye solo.

Y gracias a los primeros lectores y a los colaboradores de pago libre que hacen que "gratuito en línea" sea algo que puedo seguir haciendo.

Colofón

Compuesto desde Markdown, construido con bookgen, un pequeño pipeline puro en Rust (sin Python).

Basado en entrevistas y asistido por IA; editado y verificado a mano. Escrito sin rayas largas. Portada e ilustraciones de capítulo de las colecciones Corvid and Nature en Algorand.